COMPLEXITY OF DYCK-REACHABILITY IN DIRECTED GRAPHS

BY

ADITYA SHANKAR NARAYANAN

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2021

Urbana, Illinois

Adviser:

Professor Mahesh Viswanathan

ABSTRACT

We study the problem of Dyck-reachability in directed graphs defined as follows: given a directed graph with edges labelled by either open or close parentheses, we claim that a vertex is Dyck-reachable from another if there is a path between these two vertices such that the string described by concatenating the edge labels of the path is a member of the Dyck language, i.e., the language of balanced parentheses. We present previous works on upper bounds in Dyck-reachability in graphs and the equivalent formulation of the problem in the context of Recursive State Machines. We also present known lower bounds for the Dyck-reachability problem and the conditional reduction to Boolean Matrix Multiplication as well as the k-clique problem. Finally, we give a linear time algorithm that computes st-Dyck-reachability for graphs with bounded treewidth and using a bounded stack.

ACKNOWLEDGMENTS

First and foremost, I want to thank my adviser, Professor Mahesh Viswanthan, without whose guidance and assistance, this thesis could not have been completed. I thank Mahesh for his support and patience while I prepared this thesis, as well his insightful feedback and support during the research process.

I also want to thank Aniket Murhekar for his incredibly helpful insights and collaboration on the research required to complete this thesis.

Finally, I thank my friends and family for their endless love and support.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

In this thesis, we study the algorithmic complexity of solving the problem of Dyck-Reachability in graphs. Graphs are incredibly useful and interesting combinatorial data structures consisting of a set of *vertices* (or *nodes*) that are interconnected by a relation called *edges*. Formally, a graph is represented as a tuple of two sets - the set of vertices, and the set depicting the edge relation. For instance, $G = (V, E)$ represents a graph G that has vertices given by the elements of the set V and edges given by the relation $E \subseteq V \times V$. This means that if there is an edge between any two vertices $u, v \in V$, then there is an edge $(u, v) \in E$.

Graphs can be classified into different types based on the properties of the vertices and edges. For the purpose of this thesis, we are interested only in one type of classification of graphs: graphs can be classified as *directed* or *undirected* based on the directionality of the edges. A *directed graph*, often called a *digraph*, is a graph where the edges are oriented in a particular direction; for instance, if $(u, v) \in E$ is an edge in a directed graph G, then the vertex u denotes the starting vertex of the edge and the the vertex v denotes the ending vertex of the edge, thus indicating that the edge is oriented from u to v. In the case of an undirected graph, an edge between two vertices does not have any particular orientation and is thus a symmetric relation by definition. For instance, if $(u, v) \in E$ is an edge in the undirected graph G, then there is an edge between the vertices u and v with no particular direction or orientation. Further, $(u, v) \in E$ and $(v, u) \in E$ are equivalent edges in the undirected graph G since the edge relation E is symmetric.

In this thesis, we are interested in digraphs. In particular, we are interested in solving the problem of Dyck-Reachability in digraphs. Before we dive deeper into this problem and why it is interesting, let us review some key concepts that will help us understand this problem.

1.1 PRELIMINARIES

Definition 1.1 (Kleene Star(*))**.** Given a set V, we define

$$V_0 = \{\varepsilon\} \qquad \text{and} \qquad V_1 = V \tag{1.1}$$

where $\{\varepsilon\}$ represents the language consisting only of the empty string. We then recursively define the set V^{i+1} for all integers $i > 0$ as

$$V^{i+1} = \{wv : w \in V^i \text{ and } v \in V\} \tag{1.2}$$

That is, if V is a formal language (set of strings), then V^i is said to be the i^{th} power of the set V and is a shorthand for the concatenation of the strings in set V with itself i times. That is, V_i can be understood to be the set of all strings that can be represented as the concatenation of i strings from set V.

The definition of Kleene star operation, denoted as *, on the set V is

$$V^\star = \bigcup_{i \geq 0} V^i = V^0 \cup V^1 \cup V^2 \cup V^3 \cup V^4 \cup \dots \tag{1.3}$$

The set $V^\star$ can thus be described as the set containing the empty string along with all finite-length strings that can be generated by concatenating arbitrary elements of the set V, allowing the use of the same element multiple times. If V is either the empty set, $\emptyset$, or the singleton set containing the empty string $\{\varepsilon\}$, then $V^\star = \{\varepsilon\}$. Otherwise, for any other finite set or countably infinite set V, the set $V^\star$ is a countably infinite set. QED.

Definition 1.2 (Context-Free Language)**.** In formal language theory, a Context-Free Language (CFL) is a set of strings that can be generated by a Context-Free Grammar (CFG) or equivalently, by a pushdown automata. A Context-Free Grammar $\mathbb{G}$ is defined by the following tuple $\mathbb{G} = (N, \Sigma, P, S)$ where:

- N denotes a finite set of Non-terminal symbols, also called Variables, (usually represented by uppercase characters, e.g. $N = \{S, A, B\}$);
- Σ denotes a finite set of Terminal symbols, also called Alphabet, (usually represented by lowercase characters or other symbols, e.g. $\Sigma = \{a, b, 0, 1, \alpha, \epsilon, +, \$\}$);
- P denotes a finite set of Production rules (definition 1.3) that is given by the relation $P \subseteq N \times (N \cup \Sigma)^\star$ (where the * represents the Kleene star operation);
- S denotes the starting symbol of the grammar.

Any context-free grammar must have exactly one starting symbol $S \in N$ which must be a non-terminal. This non-terminal symbol is used to generate all the strings that can be generated by the grammar by applying the production rules from the set P (explained in definition 1.4). QED.

Given a Context-Free Grammar $\mathbb{G} = (N, \Sigma, P, S)$, we define the following.

Definition 1.3 (Production Rules)**.** A production rule in the set P is represented mathematically as a pair $(\alpha, \beta) \in P$, where $\alpha \in N$ is a non-terminal and $\beta \in (N \cup \Sigma)^\star$ is a string of non-terminals and/or terminals. Rather than using the ordered pair notation, production

rules are usually written using an arrow operator ($\rightarrow$) with α as its left hand side and β as its right hand side:

$$\alpha \rightarrow \beta$$

and is read as "α produces β". Henceforth, production rules will be represented by the arrow notation. β is allowed to be the empty string, denoted by ε. In this case, the production rule $\alpha \rightarrow \varepsilon$ is called an ε-production.

It is common (but not required) to list all right-hand sides for the same left-hand side on the same line, using | (the pipe symbol) to separate them. Rules $\alpha \rightarrow \beta_1$ and $\alpha \rightarrow \beta_2$ can hence be written as $\alpha \rightarrow \beta_1 \mid \beta_2$. In this case, β_1 and β_2 are called the first and second alternative, respectively. QED.

Definition 1.4 (Rule Application)**.** For any strings $u, v \in (N \cup \Sigma)^\star$, we say u directly yields v, written as $u \Rightarrow v$, if there exists a production rule $\alpha \rightarrow \beta \in P$ with the non-terminal symbol $\alpha \in N$, and some strings $u_1, u_2 \in (N \cup \Sigma)^\star$ such that $u = u_1 \alpha u_2$ and $v = u_1 \beta u_2$. Thus, v is a result of applying the rule $\alpha \rightarrow \beta$ to u.

For any strings $u, v \in (N \cup \Sigma)^\star$, we say u yields v or v is derived from u if there is a positive integer k and strings $u_1, \ldots, u_k \in (N \cup \Sigma)^\star$ such that $u = u_1 \Rightarrow u_2 \Rightarrow \cdots \Rightarrow u_k = v$. This relation is denoted $u \xrightarrow{\star} v$. If we have that $k \geq 2$, then the relation $u \xrightarrow{+} v$ holds. In other words, $\xrightarrow{\star}$ and $\xrightarrow{+}$ are the reflexive transitive closure (allowing a string to yield itself) and the transitive closure (requiring at least one step) of the rule-application, $\Rightarrow$, respectively. QED.

Definition 1.5 (Language of a grammar)**.** The language corresponding to a CFG $\mathbb{G}$ can be represented as $L(\mathbb{G}) = \{w \in \Sigma^\star : w \text{ is a string generated by the starting symbol of the grammar } \mathbb{G}\}$. This is formally represented as

$$L(\mathbb{G}) = \{w \in \Sigma^\star : S \xrightarrow{\star} w\}, \tag{1.4}$$

where S is the starting symbol of the grammar $\mathbb{G}$. QED.

Definition 1.6 (Context-Free Recognition)**.** Given an input string $w \in \Sigma^\star$ and a context-free language L (or equivalently, the context-free grammar $\mathbb{G}$ such that $L(\mathbb{G}) = L$), the problem of checking if w is an element of the context-free language L (i.e., if the string w can be generated by the context-free grammar $\mathbb{G}$) is called the Context-Free Recognition problem, or the Parsing problem. QED.

Definition 1.7 (Dyck Language)**.** The Dyck Language is the language corresponding to the Dyck Grammar, which is a context-free grammar used to generate strings of balanced

parentheses (called Dyck words). Formally, the Dyck Grammar given by $\mathbb{G} = (N, \Sigma, P, S)$ where $N = \{S\}$ is a set of one non-terminal symbol, S, which is also the starting symbol of the grammar; the set of terminal symbols, $\Sigma = \{(_i,)_i\} \cup \{\varepsilon\}$, where $(_i$ represents the i^{th} type of open parenthesis and $)_i$ represents the corresponding close parenthesis, and ε represents the empty string. The production rules of the grammar in the set P are as follows: For all $(_i,)_i \in \Sigma$

$$S \quad \rightarrow \quad SS \quad | \quad (_i S)_i \quad | \quad \epsilon \tag{1.5}$$

Formally, the Dyck language would then be defined as

$$L(\mathbb{G}) = \{w \in \Sigma^\star : \forall (_i,)_i \in \Sigma, \text{ any prefix of } w \text{ contains no more })_i \text{ than } (_i \text{ and, the number of })_i \text{ in } w = \text{ the number of } (_i \text{ in } w\} \tag{1.6}$$

QED.

Definition 1.8 (Path)**.** Given a graph $G = (V, E)$, a path, called π, between any 2 vertices, say from $u \in V$ to $v \in V$, is a sequence of edges, $e_1, e_2, \ldots, e_k \in E$ (for some positive integer k) such that the ending vertex of each edge, e_i, in the path is the starting vertex of the subsequent edge, e_{i+1}, in the path (for all $1 \leq i < k$), and the starting vertex of e_1 is the vertex u and the ending vertex of e_k is vertex v. We also say that the length of the path π is the number of edges in the path i.e., k. In this thesis, we formally denote the path π from vertex u to vertex v as $\pi = u \rightsquigarrow v$ and the length of the path as $|\pi| = k$. QED.

Definition 1.9 (CFL Reachability)**.** Given a directed graph $G = (V, E)$ whose edges are labeled by a character $\ell \in \Sigma$, for some alphabet Σ, and given a context-free grammar $\mathbb{G} = (N, \Sigma, P, S)$ on the same set of non-terminals Σ, a vertex $v \in V$ is said to be L-reachable from a vertex $u \in V$ if and only if there is a path called π from the vertex u to the vertex v of any length (number of edges), denoted by $\pi = u \rightsquigarrow v$, such that the label of the path obtained by concatenating the labels of the edges of the path forms a string w that is a member of the context-free language L that corresponds to the language of the grammar $\mathbb{G}$, i.e., $L = L(\mathbb{G})$. Formally,

$$v \text{ is } L\text{-reachable from } u \iff \exists \pi = u \rightsquigarrow v \in E^\star : \mathit{label}(\pi) \in L, \text{ where} L = L(\mathbb{G}) \tag{1.7}$$

QED.

Definition 1.10 (Dyck-Reachability)**.** Given a directed graph $G = (V, E)$ with edges labeled

by parentheses from the alphabet Σ, a vertex $v \in V$ is said to be Dyck-reachable from another vertex $u \in V$ if and only if there is a path from u to v (of any length) whose label reads a Dyck word (i.e., a string of well-balanced parentheses). It can be noted that Dyck-Reachability is a specific type of CFL-Reachability, where the context-free language is constrained to be the Dyck language. Alternatively, CFL-reachability is a generalization of the Dyck-Reachability problem. QED.

1.2 PROBLEM STATEMENT

There are multiple formulations of the Dyck-reachability problem in theoretical computer science. The *all-pairs reachability* problem corresponds to the problem of checking if every pair of vertices in a given digraph is Dyck-reachable from one another. The *single-source single-target Reachability*, also known as *st*-reachability, corresponds to the problem of checking if a given target vertex t is Dyck-reachable from a given source vertex s. These versions of the Dyck-reachability problem belong to the class of decision problems (i.e., the problems whose solution is of the form YES/NO or TRUE/FALSE). There are other versions that involve listing and/or counting the number of Dyck-reachable vertex pairs in a given graph or number of vertices that are Dyck-reachable from a given source vertex. However, in this thesis we are only interested in the decision problem.

There is a well-known naive dynamic programming algorithm that solves either version of the Dyck-reachability problem that uses Depth-First Search (or Breadth-First Search) to find all the paths between two vertices and check if the label of each path is recognized by the Dyck grammar. However, this solution runs in cubic time i.e., for a graph with n vertices, the naive solution takes $O(n^3)$ time to solve the Dyck-reachability problem, as the time taken by the dynamic programming algorithm (called the CYK parsing algorithm) used to check if a given string of length $|w|$ is recognized by a context-free grammar runs in $O(|w|^3)$ and in the worst case, $|w| = n$, thus dominating the runtime of the dynamic programming algorithm.

All known algorithms for solving the CFL-reachability problem follow a dynamic-programming scheme known as summarization [1][2]. Unlike context-free recognition, which has a well-known subcubic solution given by Valiant [3], CFL-reachability has not been known to have a subcubic algorithm, even in the *st*-reachability formulation of the problem. This raises the question: is this problem intrinsically cubic? The question is especially interesting in program analysis as problems like interprocedural data-flow analysis and slicing are not only solvable using CFL-reachability, but also provably as hard. Believing that the answer is "yes", researchers have sometimes attributed the "cubic bottleneck" of these problems to

the hardness of CFL-reachability[4][5].

The Dyck-Reachability problem is interesting for many reasons, including and not limited to the significance in Static analysis. Static analysis techniques obtain information about programs without running them on specific inputs. These techniques explore the program behavior for all possible inputs and all possible executions. For non-trivial programs, it is impossible to explore all the possibilities, and hence various approximations are used. A standard way to express a plethora of static analysis problems is via language reachability that generalizes graph reachability. The input consists of an underlying graph with labels on its edges from a fixed alphabet, and a language, and reachability paths between two nodes must produce strings that belong to the given language. [6]

Yet another application of Dyck reachability is in alias analysis, which has been one of the major types of static analysis and a subject of extensive study. The task is to decide whether two pointer variables may point to the same object during program execution. This problem is computationally expensive, and practically relevant results are obtained via approximations. One popular way to perform alias analysis is via points-to analysis, where two variables may be aliases of one another if their points-to sets intersect. Points-to analysis is typically formulated as a Dyck reachability problem on Symbolic Points-to Graphs (SPGs), which contain information about variables, heap objects and parameter passing due to method calls. [6]

An interesting version of Dyck-reachability is the problem in the context of bounded-treewidth graphs. In the context of programming languages, the control-flow graphs for goto-free programs for many programming languages have constant treewidth. The treewidth property has received a lot of attention in algorithm community, for NP-complete problems, combinatorial optimization problems, and even graph problems such as shortest path. In the algorithmic analysis of programming languages and verification the treewidth property has been exploited in interprocedural analysis, concurrent intraprocedural analysis, quantitative verification of finite-state graphs. One of our original contributions is to develop an algorithm with an improved upperbound to solve the *st*-reachability problem in linear time for graphs with bounded treewidth and bounded stack. Since this thesis is only concerned with Dyck-Reachability, unless specified, the reachability problem (either all-pairs or *st*) refers to the Dyck-reachability problem.

1.3 ORGANIZATION

This thesis is divided into 4 sections. In the following sections we look at certain well-known upperbounds and lowerbounds on the runtime of the algorithms used to solve the

all-pairs reachability as well as the *st*-reachability in directed graphs.

In section 2, we first examine a few key results in finding the lowerbounds on the runtime of a Dyck-reachability algorithm. We first explain the result obtained by Chatterjee et.al. in [6] that proves a reduction from the CFG recognition problem to the problem of Dyck-reachability, thus proving that if there is an efficient algorithm to compute the all-pairs Dyck reachability in a digraph, then there is an efficient algorithm to solve the CFG recognition problem (i.e., given a CFG $\mathbb{G}$ and word $w \in \Sigma^\star$, to decide if $w \in L(\mathbb{G})$). We then describe Lee's result [7] that proves that CFG recognition (and hence Dyck reachability) is BMM-hard; that is, the CFG recognition problem is as hard as Boolean Matrix Multiplication. This gives us a conditional lowerbound for Dyck reachability [6] that says that Dyck-reachability can be reduced to Boolean Matrix Multiplication, subject to some conditions (bounded treewidth graphs). Finally, we look at yet another lowerbound result given by Abboud et.al. [8] that states that CFG recognition is as hard as the k-clique problem in graph theory, thus yielding another conditional lowerbound result for Dyck-reachability.

In section 3, we look at the work done by Chaudhuri [4] in improving the upperbound of the naive solution by using a special data structure called fast sets [9], inspired by Rytter's speedup technique [10], to improve the runtime of the algorithm from $O(n^3)$ to $O(n^3/\log n)$ time. We then look at yet another algorithm that solves CFL-reachability with an additional condition (bounded stack) that leads to a faster conditional upperbound of $O(n^3/\log^2 n)$ time.

Finally, in section 4, we present our original conditional upperbound that solves the *st*-reachability problem in graphs with constant treewidth (inspired by the conditional lowerbound from Chatterjee et.al.[6]) and bounded stack (inspired by the conditional upperbound from Chaudhuri [4]) that yields an algorithm with a $O(k^2 \cdot n)$ upperbound for a graph with constant treewidth, k, and number of vertices, n.

CHAPTER 2: LOWER BOUNDS OF DYCK-REACHABILITY

The problem of Dyck reachability in graphs is a standard algorithmic formulation of various problems in static analyses. The problem has a well-known naive cubic-time solution and the best known upperbound until now is $O(n^3/\log n)$ due to [4]. Dyck reachability is also known to be 2NPDA-hard, which yields a conditional cubic lower bound subject to polynomial improvements. Chatterjee et.al. [6] prove that Dyck reachability is Boolean Matrix Multiplication (BMM)-hard. Now, since Dyck reachability is a combinatorial graph problem, techniques such as fast-matrix multiplication are unlikely to be applicable. The standard BMM-conjecture [8][7] states that there is no truly sub-cubic ($O(n^{3-\delta})$, for $\delta > 0$) combinatorial algorithm for Boolean Matrix Multiplication. Chaterjee et.al. show that in addition to the lowerbound of BMM-hardness holding for general graphs, the lowerbound also applies to graphs of constant treewidth. The lowerbound for general graphs is established by showing that the CFL parsing (or CFG recognition) problem can be reduced to Dyck-reachability in general graphs. Using the result proved by Lee [7] which reduces the problem of Boolean Matrix Multiplication to CFG recognition, it is shown that Dyck reachability in general graphs is as hard as Boolean Matrix Multiplication, and thus a conditional lower-bound for Dyck-reachability is established.

Another lowerbound obtained is by Abboud et.al. in [8] which shows a reduction from the k-clique problem to the CFG recognition problem. This result is interesting as the k-clique problem is a widely studied combinatorial problem in graph theory which is known to be NP-complete. More specifically, the result states that any improvement on Valiant's algorithm for CFG recognition [3] would lead to a breakthrough improvement in the algorithm for the k-clique problem. This result is an improvement on the result by Lee [7] which showed that any algorithm for a general CFL parsing problem with running time $O(|\mathbb{G}|n^{3-\delta})$ can be converted to subcubic algorithm for Boolean Matrix Multiplication conditioned to the requirement that the grammar size be $|\mathbb{G}| = \Omega(n^6)$; nothing was known in the case of constant size grammars.

In this chapter, we will look into these aforementioned lowerbounds for the Dyck reachability problem. The first section sketches the reduction from the CFG recognition or CFL parsing to the Dyck reachability problem. The next section explains Lee's result that shows the reduction of Boolean Matrix Multiplication to the CFG recognition problem. Finally, the last section describes the reduction of the k-clique problem to the CFG recognition problem, linking the lowerbounds obtained from the previous section together.

2.1 DYCK REACHABILITY AND CFG RECOGNITION

Chaterjee et.al. [6] show that the Dyck reachability problem in general graphs is as hard as Context-Free Language (CFL) parsing or Context-Free Grammar (CFG) recognition. Consider a Context-Free Grammar $\mathbb{G}$ in Chomsky Normal Form. A gadget graph $G^{\mathbb{G}} = (V^{\mathbb{G}}, E^{\mathbb{G}})$ is constructed for the grammar as follows: The vertex set $V^{\mathbb{G}}$ contains 2 separate nodes x and y. For every production $p_i \in \mathbb{G}$,

- If p_i is of the form $A \to a$, a vertex x_i is added to $V^{\mathbb{G}}$ and the edges (x, x_i) with a label $)_A$ and (x_i, y) with a label $(_a$ are added to $E^{\mathbb{G}}$.
- If p_i is of the form $A \to BC$, then 2 vertices x_i and y_i are added to $V^{\mathbb{G}}$ and the edges (x, x_i) with label $)_A$, (x_i, y_i) with label $(_C$ (the second terminal in the production of A) followed by the edge (y_i, y) with label $(_B$ are added to $E^{\mathbb{G}}$.

Note that the labels of the edges in the gadget graph are members of the alphabet of parentheses where the $(_i$ and $)_i$ are a matching open and close parentheses respectively.

A parse graph is constructed for the grammar $\mathbb{G}$ and an input string $s = s_1 \dots s_n$ (of length $|s| = n$). The parse graph $G_s^{\mathbb{G}} = (V_s^{\mathbb{G}}, E_s^{\mathbb{G}})$ consists of two parts: the first part is a path graph (or line graph) starting with a vertex v followed by vertices $u_0, u_1, \dots, u_n$ such that the edges are (v, u_0) with label $(_S$, (u_{i-1}, u_i) with label $)_{s_i}$ for all $1 \leq i \leq n$. The second part is n copies of the gadget graph $G^{\mathbb{G}}$ with for each u_i for $0 \leq i \leq n-1$. Finally, the edges $(u_i, x^{(i)})$ with label ε and $(y^{(i)}, u_i)$ with label ε are added where $x^{(i)}$ and $y^{(i)}$ are the 2 separate nodes x and y in the i^{th} copy of the gadget graph.

An example illustration for a grammar $\mathbb{G}$ is given in 2.1 that shows how the gadget graph and parse graph are constructed. Consider the production rules of the grammar in Figure 2.1(a) for the language of strings of the form $a^n b^n$ on the set of terminals $\Sigma = \{a, b\}$. The grammar has the set of non-terminal symbols as $N = \{S, T, A, B\}$, where S is the starting symbol of the grammar. Figure 2.1(b) shows an instance of the gadget graph constructed for this grammar. As illustrated, for the production rule $S \to TB$, there is an edge from node x to node S labelled by the close parenthesis $)_S$, followed by an edge from S to x_1 labelled by the open parenthesis $(_B$ and a subsequent edge from x_1 to y labelled by the open parenthesis, $(_T$ (as per the second rule in the construction of the gadget graph). Similarly, the edges from x to T, T to x_2, and x_2 to y are constructed and labelled accordingly for the production rule $T \to AS$. The production rule $A \to a$ follows the first rule in constructing the gadget graph, having one edge from x to A labelled by the close parenthesis $)_A$ and the edge from the node A to node y labelled by the open parenthesis $(_a$. A similar construction is observed for the rule $B \to b$. Figure 2.1(c) shows the construction of the parse graph for

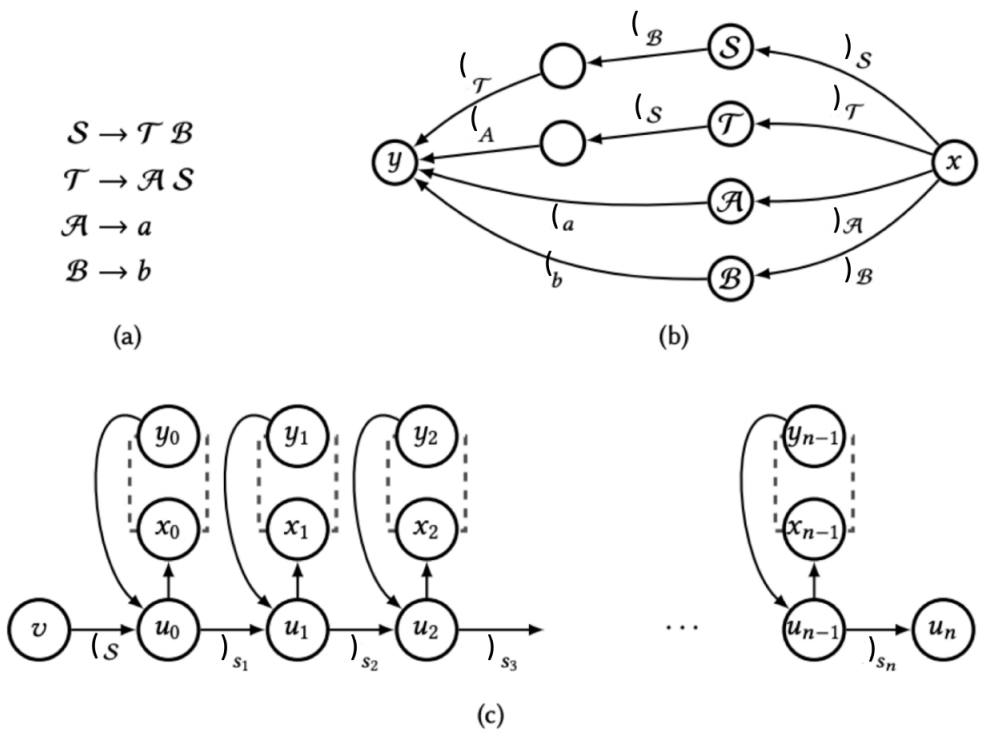

Figure 2.1: (a) The grammar $\mathbb{G}$ for the language $a^n b^n$; (b) The gadget graph $G^{\mathbb{G}}$, (c) The parse graph $G_s^{\mathbb{G}}$ for the input string $s = s_1 \dots s_n$. Adapted from Chatterjee et.al.[6]

an input string $s = s_1 \cdots s_n \in \Sigma^\star$. Here, we have the node v followed by nodes u_0, u_1, till u_n, for every character of the input string. The edges are directed from v to u_0, labelled by the open parenthesis $(_S$ corresponding to the starting symbol of the grammar, as the input string must be generated by the non-terminal S, if the string is a member of the given language. Subsequently, edges from u_i to u_{i+1} for all $0 \leq i < n$ are labelled by the closed parenthesis $)_{s_{i+1}}$ corresponding to the $(i+1)^{th}$ character of the input string s. Further, for each node u_i for $0 \leq i < n$, we have a copy of the gadget graph with an edge from u_i to the corresponding x-node of the i^{th} copy of the gadget graph and an edge from the corresponding y-node to u_i, both edges labelled by the empty string ε.

Lemma 2.1. *The node u_n is Dyck-reachable from node v in the parse graph $G_s^{\mathbb{G}}$ if and only is the input string s is generated by the grammar $\mathbb{G}$.*

Proof. Let the node u_n be Dyck-reachable from the node v in the graph $G_s^{\mathbb{G}}$. This means that there exists a path from node v to node u_n, called P, such that $\lambda(P)$ denoting the string formed by concatenating the edge labels of the path P is a word in the Dyck language i.e., a word with balanced parentheses. Clearly, for such a path to exist, every open parenthesis

$(_i$ must have a matching close parenthesis $)_i$ at the appropriate position. Any path from vertex v must start with the edge from v to u_0 labelled by $(_S$. Since all the edges on the line subgraph described by the vertices $u_0, u_1, \ldots, u_{n-1}$ are labelled by close parentheses that correspond to the terminal symbol, $)_{s_i}$, at the $i+1^{th}$ position in the input string s, the only way the path P can be a Dyck path is if every detour into the copy of the gadget graph at u_i contains an edge with the corresponding open parenthesis $(_{s_i}$. By the definition of the gadget graph, this is only possible if the the substring $s_1 \ldots s_i$ can be generated by the productions of the grammar $\mathbb{G}$. Therefore, any Dyck path from v to u_n must satisfy the condition that the string s is generated by the grammar $\mathbb{G}$. Conversely, if there is a valid derivation of the string s from the grammar $\mathbb{G}$, by pre-order traversal of the derivation tree, one can obtain a path from v to u_n that would indeed be labelled by a Dyck word as per the construction of the parse graph and the gadget graphs. Thus, the node u_n is Dyck-reachable from the node v iff s is generated by the grammar $\mathbb{G}$. QED.

Theorem 2.1 (Chatterjee et.al.[6])**.** *If there exists a combinatorial algorithm that solves the all-pairs Dyck reachability problem in time $T(n)$, where n is the number of nodes of the input graph, then there exists a combinatorial algorithm that solves the CFL parsing problem in time $O(n + T(n))$.*

From Lee's theorem (Theorem 2.3) [7], elaborated in the next section, we obtain a reduction from Boolean Matrix Multiplication to Context-Free Recognition (or Parsing). Combining the result obtained from theorem 2.3 (from the next section) with the result from theorem 2.1, we get the following conditional cubic lower-bound result for Dyck-Reachability, exploiting the BMM-conjecture [8][7] explained in the introduction of this chapter.

Corollary 2.1. (Conditional cubic lower bound) *For any fixed $\delta > 0$, if there exists a combinatorial algorithm that solves the Dyck-reachability problem in $O(n^{3-\delta})$ time, then there must exist a combinatorial algorithm that solves Boolean Matrix Multiplication in $O(n^{3-\delta/3})$ time.*

This conditional lowerbound result is one of the best known results. Currently, no unconditional lowerbound is known for the Dyck-reachability problem.

Remark 2.1. Since the size of the grammar $\mathbb{G}$ is constant, the parse graph $G_s^{\mathbb{G}}$ has a constant treewidth. This implies that the BMM-hardness of the corollary holds even for Dyck-reachability on graphs of constant treewidth.

2.2 CFG RECOGNITION AND BOOLEAN MATRIX MULTIPLICATION

This section presents Lee's reduction [7] from Boolean Matrix Multiplication to Context-Free Recognition, referred to in this section as parsing.

Definition 2.1 (Boolean Matrix Multiplication)**.** A Boolean matrix is a matrix with entries from the set $\{0, 1\}$. A Boolean matrix multiplication (BMM) algorithm takes as input two $m \times m$ Boolean matrices A and B and returns their Boolean product $A \times B$, which is the $m \times m$ Boolean matrix C whose entries are given by

$$C[i, j] = \bigvee_{k=1}^{m} (A[i, k] \wedge B[k, j]) \tag{2.1}$$

That is, $C[i, j] = 1$ if and only if there exists a number $k, 1 \leq klem$, such that $A[i, k] = B[k, j] = 1$. As noted above, the Boolean product C can be computed via standard matrix multiplication, since $C[i, j] = \Sigma_{k=1}^{m} A[i, k] \cdot B[k, j]$. QED.

Our goal in this section is to show that Boolean matrix multiplication can be efficiently reduced to c-parsing of CFGs. That is, we will describe a simple procedure that takes as input an instance of the BMM problem and converts it into an instance of the CFG parsing problem with the following property: any c-parsing algorithm run on the new parsing problem yields output from which it is easy to determine the answer to the original BMM problem. We therefore demonstrate that any c-parser can be used to solve an instance of Boolean matrix multiplication. Thus, given two $m \times m$ Boolean matrices A and B, we construct a grammar $\mathbb{G}$ and an input string w such that c-parsing w with respect to $\mathbb{G}$ yields the output $\mathbb{F}_{\mathbb{G},w}$, using which the Boolean product $C = A \times B$ can be computed.

Suppose entries $A[i, k]$ in A and $B[k, j]$ in B are both 1. If there exists some way to break up array indices into two parts so that i can be reconstructed from i_1 and i_2, j can be reconstructed from j_1 and j_2, and k can be reconstructed from k_1 and k_2, then the grammar $\mathbb{G}$ will allow the following sequence of derivations.

$$\begin{aligned} C_{i_1 j_1} &\Rightarrow A_{i_1 k_1} B_{k_1 j_1} \\ &\stackrel{\star}{\rightarrow} w_{i_2} \ldots w_{k_2+\delta} \cdot w_{k_2+\delta+1} \ldots w_{j_2+2\delta} \end{aligned} \tag{2.2}$$

where $w_{i_2} \ldots w_{k_2+\delta}$ is derived by the non-terminal $A_{i_1 k_1}$ and $w_{k_2+\delta+1} \ldots w_{j_2+2\delta}$ is derived by $B_{k_1 j_1}$, for a constant δ defined later. We observe that the non-terminal $C_{i_1 j_1}$ generates two non-terminals whose "inner" indices match (k_1), and that these two non-terminals generate substrings that lie exactly next to each other. The "inner" indices of the non-terminals constitute a check on k_1, and substring adjacency constitutes a check on k_2. Together, these two checks serve as a proof that $A[i, k] = B[k, j] = 1$, and hence that $C[i, j]$ is also 1.

Let us set a constant $d = \lceil m^{1/3} \rceil$ and $\delta = d + 2$. We choose δ to be slightly larger than d to avoid ε-productions in the grammar. We then construct an input string of length 3δ. Let i be a matrix index, $1 \leq i \leq m \leq d^3$. We define the function $f(i) = (f_1(i), f_2(i))$ as

$$f_1(i) = \lfloor i/d \rfloor \tag{2.3}$$

$$f_2(i) = (i \mod d) + 2 \tag{2.4}$$

Thus, we have $0 \leq f(i) \leq d^2$ and $2 \leq f_2(i) \leq d+1$. Since $f_1(i)$ and $f_2(i)$ are essentially the quotient and remainder of integer division of i by d, we can reconstruct i from $(f_1(i), f_2(i))$ as $i = f_1(i) \cdot d + f_2(i)$.

We now construct the grammar $\mathbb{G} = (N, \Sigma, P, S)$ with the terminal symbols $\Sigma = \{w_\ell : 1 \leq \ell \leq 3d + 6\}$. The input string $w = w_1 \dots w_{3d+6}$ is extremely simple and does not depend on the matrices A and B at all. We can therefore break the string w into 3 parts of equal length called x, y, and z, such that $w = x \cdot y \cdot z$ where $x = w_1 \dots w_{d+2}$, $y = w_{d+3} \dots w_{2d+4}$, and $z = w_{2d+5} \dots w_{3d+6}$. We notice that for any $i, 1 \leq i \leq m$, we have $2 \leq f_2(i)\ d+1$, indicating that $w_{f_2(i)}$ is in the substring x. Similarly, $d + 4 \leq f_2(i) + \delta \leq 2d + 3$, thus indicating that $w_{f_2(i)+\delta}$ is in the substring y, and since $2d + 6 \leq f_2(i) + 2\delta \leq 3d + 5$, we have that $w_{f_2(i)+2\delta}$ is in the substring z.

We start constructing the grammar $\mathbb{G}$ by starting with the set of non-terminals as $N = \{S\}$ (where S is the starting symbol) and the set of production rules $P = \emptyset$. First, we add a non-terminal W to set N that is used to generate arbitrary non-empty substrings and we add the following production rule, called "W-rules" to P:

$$W \quad \rightarrow \quad w_\ell\ W \quad | \quad w_\ell, \qquad 1 \leq \ell \leq 3d + 6 \tag{2.5}$$

Next, we add the non-terminals $A_{p,q}$ for $1 \leq p, q \leq d^2$ to encode the entries in matrix A. For every non-zero entry $A[i, j]$ in the matrix A, we add the following production rule, called "A-rules", to P:

$$A_{f_1(i),f_1(j)} \quad \rightarrow \quad w_{f_2(i)}\ W\ w_{f_2(j)+\delta} \tag{2.6}$$

We then add the non-terminals $B_{p,q}$ for $1 \leq p, q \leq d^2$ to encode the entries in matrix B. For every non-zero entry $B[i, j]$ in the matrix B, we add the following production rule, called "B-rules", to P:

$$B_{f_1(i),f_1(j)} \quad \rightarrow \quad w_{f_2(i)+\delta+1}\ W\ w_{f_2(j)+2\delta} \tag{2.7}$$

To encode the entries in matrix C, we add the non-terminals $C_{p,q}$ for $1 \leq p, q \leq d^2$. We add the following production rule, called "C-rules", to P:

$$C_{p,q} \quad \rightarrow \quad A_{p,r}\ B_{r,q}, \qquad 1 \leq p, q, r \leq d^2 \tag{2.8}$$

Finally, we complete the set of production rules by adding the following "S-rule" to P:

$$S \quad \rightarrow \quad W\ C_{p,q}\ W, \qquad 1 \leq p, q \leq d^2 \tag{2.9}$$

With this constructed grammar $\mathbb{G}$ and constructed input string w, the following theorem is obtained whose detailed proof can be found in Theorem 1 of [7].

Theorem 2.2 (Lee[7]). *For $1 \leq i, j, \leq m$, the entry $C[i, j]$ of matrix C is non-zero iff the non-terminal $C_{f_1(i),f_1(j)}$ can derive the string $w_{f_2(i)}^{f_2(j)+2\delta}$ using the C-rules.*

Let us now calculate the size of the grammar $\mathbb{G}$. The set N consists of roughly $3((d^2)^2) \approx m^{4/3}$ non-terminals. The set P contains about $6d$ W-rules and $(d^2)^2 \approx m^{4/3}$ S-rules. There are at most m^2 A-rules, since there are at most m^2 non-zero entries in matrix A. Similarly, there are at most m^2 B-rules. And lastly, there are $(d^2)^3 \approx m^2$ C-rules. Therefore, our grammar is of size $O(m^2)$. We already know that $d = \lceil m^{1/3} \rceil$ and the input string w has a length $3d + 6$. Hence, $|w| = O(m^{1/3})$.

We therefore get the key result that establishes a lowerbound on CFG parsing as follows. The detailed proof of this can be found in theorem 2 of [7].

Theorem 2.3 (Lee[7]). *Any parser P with running time $O(T(g)t(n))$ on grammars of size g and strings of length n can be converted into a BMM algorithm MP that runs in time $O(\max\{m^2, T(m^2)t(m^{1/3})\})$. In particular, if P takes time $O(gn^{3-\delta})$, then MP runs in time $O(m^{3-\delta/3})$ for some constant δ.*

2.3 CFG RECOGNITION AND K-CLIQUE

This section describes the reduction of the k-Clique problem to the CFG recognition problem following theorem 1 and section 2 of Abboud et.al. [8]. The theorem is stated as follows.

Theorem 2.4. *There must exist a Context-Free Grammar $\mathbb{G}$ of constant size such that if it can be determined whether an input string of length n can be generated by $\mathbb{G}$ in $T(n)$ time, then the k-clique problem on a graph with n vertices can be solved in $O(T(n^{k/3+1}))$ time for any $k \geq 3$. Furthermore, this reduction is combinatorial.*

Proof. Given a graph $G = (V, E)$ of n vertices, a string s of length $O(k^2 \cdot n^{k+1})$ can be constructed in linear time that encodes G. Let each vertex be associated with an integer in $[n]$ and let $\overline{v}$ denote the encoding of vertex v in binary and assume that $|\overline{v}| = 2 \log n$ for all $v \in V$. When G is clear from context, we denote the set of all k-cliques of G by C_k. Let the concatenation of sequences be denoted by $x \circ y$, and the reverse of a sequence x denoted by x^R.

We define a context-free grammar $\mathbb{G}$ of constant size, independent of the graph G or the constant k, such that the string s is a member of the language defined by $\mathbb{G}$ if and only if G contains a $3k$-clique. Let the alphabet set of this grammar be the following set containing 13 terminal symbols.

$$\Sigma = \{0, 1, \#, \$, a_{start}, a_{mid}, a_{end}, b_{start}, b_{mid}, b_{end}, c_{start}, c_{mid}, c_{end}\} \tag{2.10}$$

The *node* and *list* gadgets are defined as follows:

$$NG(v) = \#\overline{v}\# \quad \text{and} \quad LG(v) = \# \bigcirc_{u \in N(v)} (\$\overline{u}^R\$)\# \tag{2.11}$$

Consider some $t = \{v_1, \ldots, , v_k\} \in C_k$. The *clique node* and *clique list* gadgets are defined as follows:

$$CNG(t) = \bigcirc_{v \in t}(NG(v))^k \quad \text{and} \quad CLG(t) = (\bigcirc_{v \in t} LG(v))^k \tag{2.12}$$

and the main clique gadgets are

$$CG_\alpha(t) = a_{start}\ CNG(t)\ a_{mid}\ CNG(t)\ a_{end} \tag{2.13}$$

$$CG_\beta(t) = b_{start}\ CNG(t)\ b_{mid}\ CNG(t) b_{end} \tag{2.14}$$

$$CG_\gamma(t) = c_{start}\ CNG(t)\ c_{mid}\ CNG(t) c_{end} \tag{2.15}$$

Finally, the graph G is encoded as the string s as follows.

$$s = (\bigcirc_{t \in C_k} CG_\alpha(t)) \cdot (\bigcirc_{t \in C_k} CG_\beta(t)) \cdot (\bigcirc_{t \in C_k} CG_\gamma(t)) \tag{2.16}$$

The set of non-terminal symbols of the grammar $\mathbb{G}$ is

$$T = \{S, W, W', V, S_{\alpha\beta}, S_{\beta\gamma}, S_{\alpha\gamma}, S^*_{\alpha\beta}, S^*_{\beta\gamma}, S^*_{\alpha\gamma}, N_{\alpha\beta}, N_{\beta\gamma}, N_{\alpha\gamma}\}. \tag{2.17}$$

The "main" production rules in the grammar are:

$$S \rightarrow W\ a_{start}\ S_{\alpha\gamma}\ c_{end}\ W \tag{2.18}$$
$$S^*_{\alpha\beta} \rightarrow a_{end}\ W\ b_{start} \tag{2.19}$$
$$S^*_{\beta\gamma} \rightarrow b_{end}\ W\ c_{start} \tag{2.20}$$
$$S^*_{\alpha\gamma} \rightarrow a_{mid}\ S_{\alpha\beta}\ b_{mid} S_{\beta\gamma}\ c_{mid} \tag{2.21}$$

For all $xy \in \{\alpha\beta, \beta\gamma, \alpha\gamma\}$ and $\sigma \in \{0, 1\}$, we have the following "listing" production rules.

$$S_{xy} \rightarrow S^*_{xy} \mid \#N_{xy}\$V\# \tag{2.22}$$
$$N_{xy} \rightarrow \#S^{xy}\#V\$ \mid \sigma N_{xy}\sigma \tag{2.23}$$

For all $t \in \Sigma$, the "assisting" production rules are:

$$W \rightarrow tW|\varepsilon \tag{2.24}$$
$$W' \rightarrow \sigma W'|\varepsilon \tag{2.25}$$
$$V \rightarrow \$W'\$V|\varepsilon \tag{2.26}$$

Clearly, the grammar $\mathbb{G}$ has a constant size as it has 13 terminal symbols Σ, 13 non-terminal symbols T, and 38 production rules with the sum of the lengths of the production rules being 132. Now, the proof of correctness for this reduction follows by proving both the directions of the biconditional claim: Any encoding of a graph into a string will be generated by the grammar $\mathbb{G}$ iff the graph contains a $3k$-clique.

Claim 1. *If $\mathbb{G} \xrightarrow{\star} w$, then G contains a $3k$-clique.*

Proof. Since the input string w is derived by the grammar $\mathbb{G}$, specifically by the start symbol of the grammar S, the first derivation used must be the rule

$$S \xrightarrow{\star} w_1\ a_{start}\ S_{\alpha\gamma}\ c_{end}\ w_2 \tag{2.27}$$

where a_{start} appears in $CG(t_\alpha)$ and c_{start} in $CG(t_\gamma)$ for some $t_\alpha, t_\gamma \in C_k$, w_1 is the prefix of w before $CG(t_\alpha)$, and w_2 is the suffix of w after $CG(t_\gamma)$. Then we get

$$S_{\alpha\gamma} \xrightarrow{\star} CNG(t_\alpha)\ S^\star_{\alpha\gamma}\ CLG(t_\gamma) \tag{2.28}$$

by repeatedly applying the "listing" rules with xy as $\alpha\gamma$ and and finally using the rule $S_{\alpha\gamma} \rightarrow s^{\star}_{\alpha\gamma}$. It can be shown that if the above derivation is possible, then the vertices in $t_\alpha \cup t_\gamma$ form a $2k$-clique in G. Similarly, the vertices in $t_\beta \cup t_\gamma$ also form a $2k$-clique. Finally,

$$S^{\star}_{\alpha\beta} \xrightarrow{\star} a_{end}\ w_3\ b_{start} \tag{2.29}$$

$$S^{\star}_{\beta\gamma} \xrightarrow{\star} b_{end}\ w_4\ c_{start} \tag{2.30}$$

where w_3 and w_4 are substrings of w between $CG(t_\alpha)$ and $CG(t_\beta)$, and $CG(t_\beta)$ and $CG(t_\gamma)$ respectively. Therefore, combining these observations, it can be concluded that the vertices in $t_\alpha \cup t_\beta \cup t_\gamma$ form a $3k$-clique in the graph G. QED.

Claim 2. *If G contains a $3k$-clique, then $\mathbb{G} \xrightarrow{\star} w$.*

Proof. This can be proved by following the derivations in the proof of *Claim 1* with any triple $t_\alpha, t_\beta, t_\gamma \in C_k$ of k-cliques, whose union forms a $3k$-clique. QED.

Thus, given an instance of a $3k$-Clique in a graph G, we construct the string w of length $O(k^2 \cdot n^{k+1})$ in time linear to the length of the string, as described. Given an algorithm that decides if the constant-size grammar $\mathbb{G}$ can generate the input string of length n in time $T(n)$, it can be checked if $\mathbb{G} \xrightarrow{\star} w$ in time $O(T(n^{k/3+1}))$, for a constant k. By Claims 1 and 2, $\mathbb{G} \xrightarrow{\star} w$ iff the graph G contains a $3k$-clique. Hence, the theorem is proved. QED.

CHAPTER 3: DYCK-REACHABILITY AND RECURSIVE STATE MACHINES

The problem of Dyck Reachability is graphs is a subset of the more general problem of Context-Free Language (CFL) Reachability, which extends to any Context-Free Language and not just the Dyck language. In this chapter, we explore some well-known upperbounds associated with Dyck (and CFL) reachability. We first start by looking at an improvement over the naive cubic-time dynamic programming algorithm for CFL reachability or its equivalent problem, RSM (Recursive State Machine) reachability, that gives an inverse logarithmic factor of improvement by making use of Rytter's technique [10] of applying Fast sets [9][11] as shown by Chaudhuri [4]. Next, we look at Chaudhuri's algorithm for RSM reachability with bounded-stack RSMs [4], that provides inspiration for our work presented in chapter 4.

3.1 RSM PRELIMINARIES

Recursive state machines (RSMs) are finite-state-machines that call other finite-state-machines recursively [1]. It is shown in [1] that RSMs are equivalent to pushdown automata and any solution to the RSM reachability problem can be translated to a solution for pushdown automata with the same complexity, essentially implying that RSM reachability and CFL reachability are equivalent formulations of the same problem.

Definition 3.1 (Recursive State Machines)**.** A Recursive State Machine, henceforth referred to as RSM, M can be defined as a tuple $\langle M_1, \ldots, M_k \rangle$ for some k, where each $M_i = \langle Q_i, B_i, f_i En_i, Ex_i, \rightarrow_i \rangle$ is a finite-state machine or *component* comprising of the following:

- a finite set of *internal* states, Q_i,
- a finite set of *boxes*, B_i,
- mapping functions $f_i : B_i \rightarrow \{1, 2, \ldots, k\}$ that assigns a component M_i to every box,
- a finite set of entry and exit states, $En_i \subseteq Q_i$ and $Ex_i \subseteq Q_i$ respectively,
- and an edge relation $\rightarrow_i \subseteq (Q_i \cup Ret_i \backslash Ex_i) \times (Q_i \cup Call_i \backslash En_i)$, where $Call_i = \{(b, en) : b \in B_i, en \in En_{Q_i(b)}\}$ and $Ret_i = \{(b, ex) : b \in B_i, ex \in Ex_{Q_i(b)}\}$ corresponding to the set of calls and returns in M_i

No edges can start from a call state or an exit state and no edge can terminate at a return state or an entry state. Furthermore, for all distinct i and j, the $Q_i, B_i, Call_i, Ret_i$ are distinct from $Q_j, B_j, Call_j, Ret_j$. The set of all states in M is given by $V = \bigcup_i (Q_i \cup Call_i \cup Ret_i)$ and the set of all boxes in M is given by $B = \bigcup_i B_i$. The *size* of an RSM is given by the total number of states in it. QED.

Definition 3.2 (Configuration graph)**.** Let C_M denote the infinite configuration graph of an RSM M, whose nodes are given by $c = (v, w) \in V \times B^\star$ where if $w = b_1 \dots b_n$ for some $n \geq 1$, then $v \in V_{f(b_n)}$ and for all $i \in \{1, \dots, n-1\}$, $b_{i+1} \in B_{f(b_{i+1})}$, where f is the mapping function that maps the boxes to integers from 1 to k. The edges of C_M are given by satisfying the following condition: there is an edge from $c = (v, w)$ to $c' = (v', w')$ iff one of the following conditions hold:

- *Local move:* $v \in (Q_i \cup Ret_i) \backslash Ex_i$, $(v, v') \in \rightarrow_i$, and $w' = w$.
- *Call move:* $v = (b, en) \in Call_i$, $v' = en$ and $w' = w \cdot b$.
- *Return move:* $v \in Ex_i$, $v' = (b, v)$ and $w = w' \cdot b$

The string w in a configuration (v, w) represents a stack and the paths in C_M define the operational semantics of M. If v is a call state (b, en), then the RSM pushes b onto the stack and moves to the entry state en of the component $f(b)$. On reaching an exit ex, b is popped from the stack and the RSM moves to the return state (b, ex). QED.

Definition 3.3 (RSM Reachability)**.** The problem of RSM reachability is simply the problem of reachability in the configuration graph of a given RSM which is defined as follows: The state v' is reachable from v if a configuration (v', w) for a stack w is reachable from (v, ϵ) in the configuration graph. QED.

Definition 3.4 (Same Context Reachability)**.** The state v' is said to be same-context reachable from v if in the configuration graph of the RSM, (v', ϵ) is reachable from (v, ϵ), which can only happen if v and v' are in the same component. QED.

Definition 3.5 (Equivalence of RSM Reachability and CFL Reachability)**.** From Yannakakis [12], it is known that the graph problem of CFL reachability is equivalent to the problem of RSM reachability.

Let S be a directed graph whose edges are labeled by elements from the alphabet Σ, and let L be a context-free language over Σ. The RSM reachability formulation equivalent to the L-reachability problem in S is given as follows: Let $G = (V, E)$ be a graph whose nodes are states of the RSM M. For all edges $(u, v) \in E$, S has an edge from u to v labeled by a

symbol a. For every call state (b, en) in the M, S has an edge labeled $({}_b$ from (b, en) to en. For every exit state ex and return state (b, ex) in M, there is an edge in S from ex to (b, ex) labeled by $)_b$. The state v is reachable from the state u in M iff v is L-reachable from u in S, where L is given by the Dyck grammar i.e.,

$$S \quad \rightarrow \quad SS \quad | \quad ({}_bS)_b \quad | \quad ({}_bS \quad | \quad a. \tag{3.1}$$

QED.

3.2 RSM REACHABILITY

All known algorithms for RSM reachability are cubic and are based on the summarization algorithm. However, using the speedup techniques of Rytter [10], Chaudhuri [4] was able to develop a $O(n^3/\log n)$ time solution by computing reachability via a sequence of operations on sets of states, each represented as a fast set [9][11]. QED.

Definition 3.6 (Fast Sets)**.** Let U denote a universe of n elements of which all sets will be subsets. The fast set data structure supports the following operations:

- *Insertion* of a value into the set.
- *Set difference*: Given sets A and B, return a list $\mathit{Diff}(A, B)$ consisting of the elements of the set $A \backslash B$.
- *Assign-union*: Given sets A and B, assign the set $A \leftarrow A \cup B$.

For an architecture with a word size $p = \theta(\log n)$, the fast set is represented as an n bit vector, broken into $\lceil n/p \rceil$ words. Setting any bit in a word is a unit cost operation and thus the insertion operation can be done in $O(1)$ time. The set difference operation, implemented as *Diff*, can be done in $O(|C| + n/p)$ where $C \leftarrow \mathit{Diff}(A, B)$. The assign-union operation can also be implemented in $O(n/p)$ time. Unit-cost operations can be implemented as a table lookup and so for $p = \lceil \log n/2 \rceil$, a table of size $O(2^p \cdot 2^p) = O(n)$ possible inputs can be created in $O(pn)$ time and $O(n)$ space. QED.

Given an RSM $M = \langle M_1, \ldots, M_k \rangle$ with state set V, box set B, edge relation $\rightarrow \subseteq V \times V$, and a map $f : B \rightarrow \{1, \ldots, k\}$ that assigns components to boxes, Chaudhuri's algorithm determines same-context reachability by building a relation $H^s \subseteq V \times V$ defined as follows:

- if $u = v$ or $u \rightarrow v$, then $(u, v) \in H^s$
- if $(u, v') \in H^s$ and $(v', v) \in H^s$, then $(u, v) \in H^s$

- if $(u, v) \in H^s$ and u is an entry and v is an exit in some component, then $\forall b \in B, (b, u), (b, v) \in V$, we have $((b, u), (b, v)) \in H^s$.

H^s can be computed using a fixed point computation despite being recursively defined. Using this, the relation H is computed as

$$H = \rightarrow \cup \{((b, en), (b, ex)) \in H^s : b \in B, en \in En_{f(b)}, ex \in Ex_{f(b)}\} \quad (3.2)$$

$$\cup \{((b, en), en) : en \in En_{f(b)}\} \quad (3.3)$$

Once H is computed, the transitive closure of H, given by $H^\star$ is computed. From Alur et.al.[1] and Bouajjani et.al.[2], the following lemma is obtained.

Lemma 3.1. *For states u and v of M, v is reachable from u iff $(u, v) \in H^\star$. Also, v is same-context reachable from u iff $(u, v) \in H^s$.*

This gives the first subcubic algorithm for RSM reachability called Reachability in [4].

Theorem 3.1. *The algorithm* Reachability *solves the all-pairs reachability and same-context-reachability problems for an RSM with n states in $O(n^3/\log n)$ time and $O(n^2)$ space.*

From this and the equivalence of RSM reachability and CFL reachability, we get the following theorem.

Theorem 3.2. *The algorithm* CFL-Reachability *solves the all-pairs CFL-reachability problem for a fixed-sized grammar and a graph with n vertices in $O(n^3/\log n)$ time and $O(n^2)$ space.*

The algorithm 3.1 gives the optimized CFL-reachability algorithm that uses fast sets to store the relation H^s and use table lookups, thus offering an inverse logarithmic factor of speedup over the original cubic time algorithm as given by Melski and Rep [5]. This modification can be seen in the lines

$$\text{for } u' \in Col(u) \backslash Col(v) \qquad \text{and} \qquad \text{for } v' \in Row(u) \backslash Row(v) \quad (3.4)$$

The fast set implementation cost for these loop in a given iteration of the main loop is $O(n/\log n + \sigma)$, where σ is the number of new insertions into H^s. Since the number of insertions into H^s is $O(n^2)$, the total cost during a complete run of the algorithm is $O(n^3/\log n)$.

Algorithm 3.1: CFL-Reachability

```
input : A labeled digraph S with n nodes, Constant-sized Grammar G with start
        symbol A
output: H^s
begin
    W ← {(u, A, v) : u -a-> v ∈ S, (A → a) ∈ G} ;
    W ← W ∪ {(u, A, u) : (A → ϵ) ∈ G} ;
    H^s ← W while W ≠ ∅ do
        (u, B, v) ← pop(W) ;
        for each production A → B do
            Insert (u, B, v) into H^s, W ;
        end
        for each production A → CB do
            for u' ∈ Col(u)\Col(v) do
                Insert (u', A, v) into H^s, W ;
            end
        end
        for each production A → BC do
            for v' ∈ Row(v)\Row(u) do
                Insert (u, A, v') into H^s, W ;
            end
        end
    end
end
```

3.3 BOUNDED STACK RSM REACHABILITY

Chaudhuri [4] gives an $O(n^3/\log^2 n)$ time algorithm to compute graph transitive closure and extend its application into RSMs by combining Tarjan's algorithm as used by Purdom [13] to compute graph transitive closure, with Rytter's [10] fast-set [9] based speedup technique. While other subcubic algorithms are knows for computing graph transitive closure, Chaudhuri's algorithm is the first one so far to be based on graph traversal and yet have a subcubic runtime.

Definition 3.7 (Bounded Stack RSM)**.** The class of bounded-stack RSMs consists of RSMs M where every call (b, en) is unreachable from the entry state en. The stack of an RSM grows along an edge from a call state to its corresponding entry state. Therefore, a bounded stack RSM forbids infinite recursive loops, ensuring that in any path in the configuration

graph of M starting with a configuration (v, ϵ), the height of the stack stays bounded by a constant d. QED.

Theorem 3.3. *The algorithm* Same-Context-Bounded-Stack-Reachability *computes all-pairs reachability in a bounded-stack RSM of size n in $O(n^3/\log^2 n)$ time and $O(n^{5/2}/\log n)$ space.*

The algorithm 3.3 is able to perform this task as described. We can get a high level understanding of the working of this algorithm by looking at the following:

Definition 3.8 (Summary graph)**.** Let M be an RSM. We view the relation H defined as a graph called the summary graph of M. The edges of H are classified as follows:

- *Call Edges*: Edges of the form $((b, en), en)$, where b is a box and en is an entry state in $f(b)$;
- *Summary Edges*: Edges of the form $((b, en), (b, ex))$, where b is a box, en is an entry, and ex an exit in $f(b)$;
- *Local Edges*: Edges that are also present in M.

QED.

Note that a state v is same-context reachable from a state u iff there is a $u \rightsquigarrow v$ path in H consisting of only local and summary edges. Let the set of states same-context reachable from u be denoted by $H^s(u)$. While the call and local edges of H are specified directly by M, we need to determine reachability between entries and exits in order to identify the summary edges. A search algorithm is employed to compute reachability in H. When an exit ex is same-context reachable from en, the corresponding summary edge $((b, en), (b, ex))$ is added to the graph. The algorithm must explore the newly added edges along with the original edges in the graph.

Let us assume that M is a bounded-stack RSM. Consider any call (b, en) in the summary graph H. Because M is bounded-stack, this state is unreachable from the state en. Hence, (b, en) and en are not in the same strongly connected component (SCC) in H, and a call edge is always between two SCCs. It can be argued that t all summary edges in H may be discovered using a variant of depth-first graph search (DFS). This is shown by Chaudhuri in [4].

Let $Reach(v)$ denote the set of nodes reachable from a node v in a graph. It can be observed that for any two nodes v_1 and v_2 in the same SCC of a graph, we have $Reach(v1) = Reach(v2)$. Thus, it is sufficient to compute the set *Reach* for a single representative node per SCC. Another main idea in the algorithm is based on a property of Tarjan's algorithm.

Algorithm 3.2: VISIT

```
input : A state u of a bounded-stack RSM
begin
    Visited ← Visited ∪ {u} ;
    push(u, L) ;
    dfsnum(u), low(u) ← height(L) ;
    Out(u), H^s(u) ← ∅ ;
    rep(u) ←⊥ ;
    if u is an internal state then
        Next(u) ← {v : u → v} ;
    else if u is a call state (b, en) then
        Next(u) ← {en} ;
    else
        Next(u) ← ∅ ;
    end
    for v ∈ Next(u) do
        if v ∉ Visited then
            VISIT(u) ;
        end
        if v ∈ Done then
            if u = (b, en) is a call and v = en then
                for ex ∈ H^s(en) do
                    Next(u) ← Next(u) ∪ {(b, ex)} ;
                end
            end
            else
                Out(u) ← Out(u) ∪ {v} ;
            end
        end
        else
            low(u) ← min{low(u), low(v)} ;
        end
    end
    if low(u) = dfsnum(u) then
        repeat
            v ← Pop(L) ;
            Done ← Done ∪ {v} ;
            H^s(u) ← H^s(u) ∪ {v} ;
            Out(u) ← Out(u) ∪ Out(v) ;
            rep(v) ← u;
        until v = u;
        H^s(u) ← H^s(u) ∪ ⋃_{v∈Out(u)} H^s(rep(v)) ;
    end
end
```

Algorithm 3.3: SAME-CONTEXT-BOUNDED-STACK-REACHABILITY

```
input  : A RSM with state u and a bounded stack
output: Nodes that are same-context reachable from u
begin
    Visited ← ∅ ;
    Done ← ∅ ;
    for each node u do
        if u ∉ Visited then
            VISIT(u)
        end
    end
end
```

Definition 3.9 (Condensation Graph)**.** The condensation graph $\hat{G}$ of a given graph G is defined as follows:

- the nodes of $\hat{G}$ are the SCCs of G;
- if, for nodes S_1 and S_2 of $\hat{G}$, G has nodes $u \in S_1$, $v \in S_2$ such that there is an edge from u to v in G, then $\hat{G}$ has an edge from S_1 to S_2.

Now, when running on a graph G, Tarjan's algorithm outputs the nodes of $\hat{G}$ in a bottom-up topological order as a result of the DFS done on G. Furthermore, since the condensation graph of any graph is acyclic, for every node S in the condensation graph, the set of nodes $Reach(S)$ reachable from that SCC, defined as:

$$Reach(s) = \bigcup_{u \in S} Reach(u) \tag{3.5}$$

QED.

Suppose we are only interested in same-context reachability, we apply the transitive closure algorithm to the graph H after modifying it in the two following ways. First, we ensure that the sets $Reach(u)$, for a state u, only contain descendants of u reachable via local and summary edges (this requires a trivial modification of the algorithm). Now, consider a call (b, en) in a summary graph H. This means that the call edge $((b, en), en)$ is an edge in the condensation graph $\hat{H}$. Thus, the set $Reach(S_{en})$, where S_{en} is the SCC of en, is known by the time the transitive closure algorithm is done exploring this edge. Now, all the summary edges from (b, en) can be constructed and added as outgoing edges from (b, en),

viewing them as normal edges appearing after the call-edge in the order of exploration. The set $Reach(S_{(b,en)})$ can now be computed. By the time the above algorithm terminates, $Reach(S_u) = H^s(u)$ for each state u i.e., we have determined all-pairs same-context reachability in the RSM. To determine all-pairs reachability, we simply insert the call edges into the summary graph, and compute its transitive closure. The runtime analysis of algorithm 3.3 involves careful consideration of the use of fast-sets and a cache speedup, which can be found in section 4.1 and 4.2 of Chaudhari [4]. Thus, it is possible to obtain an upperbound of $O(n^3/\log^2 n)$ time for a bounded-stack RSM, and equivalently perform CFL reachability on a bounded-stack graph with the same upperbound.

CHAPTER 4: DYCK-REACHABILITY IN CONSTANT TREEWIDTH GRAPHS

From chapter 2, we saw an interesting result about a conditional lowerbound (Corollary 2.1) that showed that Dyck-reachability is as hard as Boolean Matrix Multiplication as proved by Chatterjee et.al. in [6]. From Remark 2.1 in chapter 2, we learned that Dyck-reachability in constant-treewidth graphs is also BMM-hard. We then saw in chapter 3, a conditional upperbound of $O(n^3/\log^2 n)$ time for same-context-reachability in bounded stack RSMs as proved by Chaudhuri in [4], that can be extended to same-context-reachability in bounded-stack graphs as well. Combining these two ideas, we decided to examine if there is a possibility of improving the upperbound for single-source-single-target Dyck reachability (or *st*-reachability) in graphs with a constant (or bounded) treewidth and a bounded stack. In this section, we present an $O(k^2 n)$ algorithm for this problem in a graph with constant treewidth k and n vertices. We consider the tree-decomposition of a graph G with a bounded stack and constant treewidth and we show that we can inductively and efficiently maintain a relation at each node of the tree decomposition that stores the same-context-reachability information for every pair of vertices given by the bag stored at each node of the tree. This relation can be computed bottom-up and efficiently using a fixed point computation.

4.1 PRELIMINARIES

Consider a graph G with n vertices and directed edges, each edge being labelled by an element from the alphabet of a Dyck Language. From [14], we utilize the following terminologies regarding tree decomposition and the treewidth of a graph.

Definition 4.1. (Tree Decomposition) A tree decomposition of a graph $G = (V, E)$ is a pair (T, β) where T is a tree and β is a family of subsets of the vertex set (referred to as *bags* in the following sections) V given by $\beta = \{B(t)\}_{t \in T}$ which satisfy the following properties:

- **(TD1)**: $\bigcup_{t \in T} B(t) = V$, i.e., the union of the family of subsets β is the same as the vertex set V of the graph G.
- **(TD2)**: $\forall uv \in E, \exists t \in T$ with $u, v \in B(t)$, i.e., for every edge in the graph G, the end vertices of the edge must be elements of some subset in the family β.
- **(TD3)**: $\forall t_1, t_2, t_3 \in T$, if $t_2 \in t_1 T t_3$, then $B(t_1) \cap B(t_3) \subseteq B(t_2)$, i.e., the intersection of the vertex subset associated with the children of a node (of T) is a subset of the vertex subset associated with the node itself.

QED.

The first two conditions, TD1 and TD2 indicate that the graph G is the union of the subgraphs induced by $B(t)$, denoted as $G(t)$, for every $t \in T$. The third condition, TD3, indicates that the parts of the tree decomposition is organized roughly like a tree.

Definition 4.2. (Treewidth) The width of a tree decomposition (T, β) of a graph G is given by

$$\max_{t \in T}(|B(t)| - 1) \tag{4.1}$$

The treewidth k of the graph G is the minimum width of all possible tree decompositions of G. QED.

We exploit the constant-treewidth property of the graph G, that allows us to consider a binary tree-decomposition of the graph where every node in the tree is either a leaf, or has one child whose bag has at most one more or one less vertex than the bag of the node.

We observe the following properties about connectivity and the tree decomposition of graphs.

Theorem 4.1. *Given a graph $G = (V, E)$ of n vertices and a constant treewidth $k = O(1)$, a tree decomposition of the graph T containing $O(n)$ nodes, each node containing a bag of k vertices from the graph G, height $O(\log n)$ and width $O(k) = O(1)$ can be constructed in $O(n)$ time.* [15]

Lemma 4.1. *Consider a graph $G = (V, E)$, its tree decomposition T, and a bag B of T. Let $(C_i)_i$ be the components of T created by removing B from T, and let V_i be the set of vertices that appear in the bags of component C_i . For every i, j, nodes $u \in V_i$, $v \in V_j$ and path $P : u \rightsquigarrow v$, we have that $P \cap B \neq \emptyset$ (i.e., all paths between u and v must go through some node in B).* [16]

4.2 *ST*-DYCK-REACHABILITY IN DIRECTED GRAPHS WITH CONSTANT TREEWIDTH AND BOUNDED STACK DEPTH

Given a graph G with a constant treewidth k, let T be the (binary) tree decomposition of G. For every node $n \in T$, let $B(n)$ denote the bag of vertices (of size k) labelling the node n and let $G(n)$ be the subgraph induced by the vertices labelling the nodes in the subtree rooted at n. Then, for every pair of vertices u and v in $B(n)$, we define the relation $R_n(u, v)$ as

$$R_n(u,v) = \{(s_1,s_2) | (u,s_1) \rightsquigarrow (v,s_2) \in G(n)\} \tag{4.2}$$

where s_1 and s_2 denote the stack configuration at vertex u and v respectively. Let the stack alphabet be represented by Γ and let the stack depth be bounded by a constant d. Due to the constant-treewidth property of G, a binary tree-decomposition T can be obtained such that every node in the tree T is either a leaf, or has one child whose bag has at most one more or one less vertex than the bag of the node, or has two children with the bags identical to the node itself. We therefore examine the last two cases (with one or more children) to see if the relation R_n can be computed for a node $n \in T$ by using the relation computed for the child/children of n.

Lemma 4.2. *If a node $n \in T$ has one child $n' \in T$ such that $B(n')$ has one more vertex than $B(n)$, then $\forall u, v \in B(n), R_n(u,v) = R_{n'}(u,v)$.*

Proof. Let $w \in B(n') \backslash B(n)$ be the extra vertex. If $B(n) \subset B(n')$ such that $|B(n') \backslash B(n)| = 1$, then $G(n)$ is a proper subgraph of $G(n')$ as the w and all the edges incident on it are in $G(n')$ and not in $G(n)$. Therefore, for all pairs of vertices in $u, v \in B(n)$, if $u \rightsquigarrow v$ is a path in $G(n')$, then $u \rightsquigarrow v$ is a path in $G(n)$ as well with identical stack configurations at u and v. Hence, $\forall u, v \in B(n), R_n(u,v) = R_{n'}(u,v)$. QED.

Lemma 4.3. *If the node $n \in T$ has only one child $n' \in T$ such that $B(n)$ has one more vertex than $B(n')$, then R_n can be computed using $R_{n'}$ in $O(k^2|\Gamma|^{2d})$ time.*

Proof. Since $B(n') \subset B(n)$ and $|B(n) \backslash B(n')| = 1$, let $w \in B(n) \backslash B(n')$ be the extra vertex. To compute the relation R_n from $R_{n'}$, we need to account for the extra edges contributed by w. We define $R_n^i(u,v)$ to be the relation on the set Γ^d such that there is a path from u with stack s to v with stack s' with at most i occurrences of the vertex w. Formally,

$$R_n^0(u,v) = R_{n'}(u,v) \tag{4.3}$$

$$\begin{aligned} R_n^{i+1}(u,v) = R_n^i(u,v) \cup \{(s,s') | \quad & \exists q, q' \in B(n), \exists a, a' \in \Gamma, \exists s_1 \in \Gamma^d, && (4.4)\\ & (s,s_1) \in R_n^i(u,q), && (4.5)\\ & q \xrightarrow{a} w, w \xrightarrow{a'} q', && (4.6)\\ & (a'(a(s_1)), s') \in R_n^0(q',v)\} && (4.7) \end{aligned}$$

We claim that if there is a path Π from u to v in $G(n)$ with at least $i+1$ w's, then the stack configurations at the start and end of Π, $(s,s') \in R_n^{i+1}$. Also, the vertices immediately

preceding and succeeding each w in Π are in $B(n)$. This claim can be verified inductively.

If $\Pi \in G(n)$ has no occurrence of w, then the path from u to v is free of the extra vertex in $G(n)$, so all the vertices of $\Pi \in G(n')$. This would mean that every $u \rightsquigarrow v$ path in $G(n)$ with 0 occurrences of w is also in $G(n')$. This means that $\forall u, v \in B(n'), R_n^0(u, v) \subseteq R_{n'}(u, v)$. Also, since $B(n') \subset B(n)$, every $u \rightsquigarrow v$ path in $G(n')$ is a $u \rightsquigarrow v$ path in $G(n)$ with 0 occurrences of the vertex w. This means that $\forall u, v \in B(n'), R_{n'}(u, v) \subseteq R_n^0(u, v)$. Therefore, we have $\forall u, v \in B(n'), R_n^0(u, v) = R_{n'}(u, v)$. Thus, the base case holds.

Assume that if there is a $u \rightsquigarrow v$ path in $G(n)$ with at least i occurrences of the vertex w, then the respective stack configurations $(s, s') \in R_n^i(u, v)$ for all $i = 0, \ldots, m$ for some positive integer m. Now, consider an arbitrary $u \rightsquigarrow v$ path in $G(n)$, Π, with $m+1$ occurrences of the vertex w. We can decompose the path Π as $\pi_1 \cdot w \cdot \pi_2$ such that π_1 is a path from u to a vertex q (that has an edge to w with label a) with m occurrences of w and π_2 is a path from q' (that has an edge to w with label a') to v with no occurrence of w. By our hypothesis, the start and end stack configurations of π_1, $(s, s_1) \in R_n^m(u, q)$. Also, the stack configuration at the start of π_2 can be obtained by pushing the labels of $q \xrightarrow{a} w$ and $w \xrightarrow{a'} q'$ successively onto the end stack configuration of π_1, i.e., s_1. Also, since $q' \rightsquigarrow v$ has no occurrences of w, by the induction hypothesis, the start and end stack configurations of π_2, $(a'(a(s_1)), s') \in R_n^0(q', v)$. Therefore, the start and end stack configurations of any $u \rightsquigarrow v$ path in $G(n)$ with $m + 1$ occurrences of w will be an element of $R_n^{m'}(u, v)$ by the recursive definition. Hence, the claim inductively holds.

We observe that the size of the recursively defined relation is bounded by $O(|\Gamma|^{2d})$. Therefore, using this fixed point computation, we can compute the relation R_n for all pairs of vertices in $B(n)$ is $O(k^2|\Gamma|^{2d})$ as $|B(n)| = k$, the constant treewidth of the graph G. QED.

Lemma 4.4. *If the node $n \in T$ has 2 children n_1 and n_2 such that $B(n_1) = B(n_2) = B(n)$, then the relation R_n can be computed using R_{n_1} and R_{n_2} in $O(k^2|\Gamma|^{2d})$ time.*

Proof. Since the bags at the vertex n and its children n_1 and n_2 are identical, the relation R_n is essentially computed by melding the relations from both the subtrees rooted at n_1 and n_2 which are R_{n_1} and R_{n_2} respectively. We observe that any $u \rightsquigarrow v$ path in $G(n)$ is either entirely present in $G(n_1) \backslash B(n)$ or $G(n_2) \backslash B(n)$ or is composed of smaller paths alternating between the aforementioned sets with the links passing through vertices in the set $B(n)$. Therefore, we define $R_n^i(u, v)$ to be a relation on the set $|\Gamma|^d$ to be the relation of start and end stack configurations of $u \rightsquigarrow v$ paths in $G(n)$ such that the path can be decomposed as $u \cdot \pi_1 \cdot u_1 \cdot \pi_2 \cdot u_2 \cdot \ldots \cdot u_i \cdot \pi_{i+1} \cdot v$ where $u_1, u_2, \ldots, u_i$ are i vertices in $B(n)$ and wlog $\pi_1, \pi_3, \ldots$ are paths that are entirely in $G(n_1) \backslash B(n)$ and $\pi_2, \pi_4, \ldots$ are paths that are entirely

in $G(n_2)\backslash B(n)$. In other words, we say that the $u \rightsquigarrow v$ path **switches** i times between the two sets, $G(n_1)\backslash B(n)$ and $G(n_2)\backslash B(n)$, from lemma 4.2. Formally,

$$R_n^0(u,v) = R_{n_1}(u,v) \cup R_{n_2}(u,v) \tag{4.8}$$

$$R_n^{i+1}(u,v) = R_n^i(u,v) \cup \{(s,s') | \exists w \in B(n), s_1 \in \Gamma^d, \quad (s,s_1) \in R_n^i(u,w), \tag{4.9}$$

$$(s_1,s') \in R_n^0(w,v)\} \tag{4.10}$$

We claim that if there exists a path Π from u to v in $G(n)$, with at least i switches between $G(n_1)\backslash B(n)$ and $G(n_2)\backslash B(n)$ (wlog), then $\Pi \in R_n^i$. This claim follows inductively.

If we consider paths from $u \rightsquigarrow v$ that do not switch between either set, then it must mean that the path is of the form $u \cdot \pi_1 \cdot v$ where $\pi_1 \in G(n_1)\backslash B(n)$ or $\pi_1 \in G(n_2)\backslash B(n)$. Therefore, the start and end configurations of these paths are in either $R_{n_1}(u,v)$ or $R_{n_2}(u,v)$ respectively. Therefore, the stack configurations of the $u \rightsquigarrow v$ path with 0 switches are present in $R_{n_1}(u,v)$ or $R_{n_2}(u,v)$ and so $R_n^0(u,v) \subseteq R_{n_1}(u,v) \cup R_{n_2}(u,v)$. Now, consider and arbitrary $(s,s') \in R_{n_1}(u,v) \cup R_{n_2}(u,v)$. These stack configurations correspond to a path π which is entirely present in either $G(n_1)\backslash B(n)$ or $G(n_2)\backslash B(n)$. Therefore, such a path π would be a $u \rightsquigarrow v$ path with 0 switches between the sets. Hence, $(s,s') \in R_n^0(u,v)$ by definition, which means that $R_{n_1}(u,v) \cup R_{n_2}(u,v) \subseteq R_n^0(u,v)$. Therefore, $R_n^0(u,v) = R_{n_1}(u,v) \cup R_{n_2}(u,v)$ and so the base case holds.

Assume that for all positive integers $i = 1, \ldots, m$, the stack configurations of any $u \rightsquigarrow v$ path with at least i switches is in the relation $R_n^i(u,v)$. Then $R_n^{m+1}(u,v)$ additionally contains the stack configurations of all $u \rightsquigarrow v$ paths with $m+1$ switches, which can be decomposed as $u \cdot \pi \cdot w \cdot \pi' \cdot v$ such that the path $u \cdot \pi \cdot w$ contains m switches and the last vertex of π is in $G(n_1)\backslash B(n)$ (wlog), $w \in B(n)$, and π' is entirely in either $G(n_2)\backslash B(n)$. By the induction hypothesis, we can conclude that $u \cdot \pi \cdot w$ with m switches having a stack configuration of s at u and s_1 at w will satisfy $(s,s_1) \in R_n^m(u,w)$. Also, since π_2 is entirely in $G(n_2)\backslash B(n)$ (wlog) with a stack configuration of s' at v, we can say that the start and end configurations of the path $w \cdot \pi_2 \cdot v$ given by (s_1,s') is an element of $R_{n_2}(w,v)$ and can also be viewed as a $w \rightsquigarrow v$ path with 0 switches. So we have $(s_1,s) \in R_n^0(w,v)$. Therefore, by the recursive definition, the stack configuration of any $u \rightsquigarrow v$ path with $m+1$ switches will be present in $R_n^{m+1}(u,v)$. Hence, the claim inductively holds.

We observe that the size of the recursively defined relation is bounded by $O(|\Gamma|^{2d})$. Therefore, using this fixed point computation we can compute the relation R_n for all pairs of vertices in $B(n)$ ($|B(n)| = k$), thus giving a runtime of $O(k^2|\Gamma|^{2d})$. QED.

Lemma 4.5. *For a node n in the tree decomposition T of a graph G with a constant treewidth k, stack alphabet Γ, and a constant bound d on the stack depth, the relation R_n can be computed for all pairs of vertices $u, v \in B(n)$ in $O(k^2|\Gamma|^{2d})$ time.*

Proof. Since G has a constant treewidth, a node n in the tree decomposition of G, given by T, is either a leaf or an internal node with at most 2 children [14]. If n is a leaf, then R_n can be computed for every pair of vertices in $B(n)$ by brute-force in $O(k^2)$ time. If n has one child, n', then either $B(n')$ has one more node than $B(n)$ or vice versa. In the first case, we know from lemma that R_n is a subset of $R_{n'}$ and can be computed in $O(k^2)$ time. In the second case, we know from lemma that R_n can be computed from $R_{n'}$ in $O(k^2|\Gamma|^{2d})$ time. Finally, if n has two children, n_1 and n_2, then the bags of vertices at nodes n, n_1, and n_2 are identical. Thus, R_n can be computed using R_{n_1} and R_{n_2} for all pairs of vertices in $B(n)$ in $O(k^2|\Gamma|^{2d})$ time, from lemma. Therefore, R_n can be computed in at most $O(k^2|\Gamma|^{2d})$ time. QED.

Therefore, we can compute the tree-decomposition T of a given bounded-stack (bounded by d) bounded-treewidth (bounded by k) graph G with n vertices in $O(n)$ time (theorem 4.1) and we can compute the relation R_n for all the leaves by brute-force in $O(k^2)$ time. Further, the relation R_n can be computed for all nodes of T in a bottom-up manner by using the fixed-point computations from lemmas 4.2, 4.3, 4.4. Once the relation R is computed for all the nodes of tree-decomposition, we can query the relation at the root node of T (denoted by T itself) to find out the same-context reachability of any two vertices s, t in the graph G by checking if $R_T(s, t)$ contains the stack configurations (s, s) for any $s \in \Gamma^\star \cup \Gamma^d$. Further, we can check if t is Dyck-reachable from s by checking if *(empty, empty)*$\in R_T(s, t)$. From lemma 4.5, we know that R_n for a node $n \in T$ can computed in $O(k^2|\Gamma|^{2d})$ time. Thus, for the relation can be computed for the root node in at most $n \cdot O(k^2|\Gamma|^{2d})$. Querying the relation can be assumed to take a constant time. Therefore, we have that *st*-reachability can be computed in $O(k^2|\Gamma|^{2d}n)$ time. We therefore get the following theorem:

Theorem 4.2. *For a graph G with a constant treewidth k and a bounded stack depth d, st-Dyck-reachability can be computed in $O(k^2|\Gamma|^{2d}n)$ time, where n is the number of vertices in G.*

It can be noted that for a bounded-stack bounded-treewidth graph, the treewidth k, the stack size $|\Gamma|$, and the stack depth d are all constants, thus allowing for *st*-Dyck-reachability to be computed in $O(n)$ time.

4.3 FUTURE WORK

Some of the future directions that can be explored could attempt to remove the dependence of the bounded-stack constraint on the graphs. A hypothesis is that for a bounded-treewidth graph with edge labels consisting of only one type of parenthesis, the following property holds: If there exists a Dyck-path (of any length) from vertex u to vertex v in a bounded-treewidth graph G such that the edge labels are only from the set of terminals $\Sigma = \{(,)\}$, and if G has n vertices, then there must exist a Dyck-path from u to v or length at most n. If this property holds, then an $O(n)$ bound can be established on the stack depth, which may remove the bounded-stack constraint. Furthermore, in the single parenthesis Dyck-reachability problem, the stack can be viewed as a 1-counter machine. Using the properties of a 1-counter machine, there may be a potential for a speedup on the runtime of single parenthesis Dyck-reachability.

REFERENCES

[1] R. Alur, K. Etessami, and M. Yannakakis, "Analysis of recursive state machines," in *Computer Aided Verification*, G. Berry, H. Comon, and A. Finkel, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 207–220.

[2] A. Bouajjani, J. Esparza, and O. Maler, "Reachability analysis of pushdown automata: Application to model-checking," in *CONCUR '97: Concurrency Theory*, A. Mazurkiewicz and J. Winkowski, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 135–150.

[3] L. G. Valiant, "General context-free recognition in less than cubic time," *Journal of Computer and System Sciences*, vol. 10, no. 2, pp. 308–315, 1975. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0022000075800468

[4] S. Chaudhuri, "Subcubic algorithms for recursive state machines," in *In POPL*, 2008, pp. 159–169.

[5] D. Melski and T. Reps, "Interconvertibility of a class of set constraints and context-free-language reachability," *TCS*, vol. 248, 1999.

[6] K. Chatterjee, B. Choudhary, and A. Pavlogiannis, "Optimal dyck reachability for data-dependence and alias analysis," *Proceedings of the ACM on Programming Languages*, vol. 2, pp. 1–30, 12 2017.

[7] L. Lee, "Fast context-free grammar parsing requires fast boolean matrix multiplication," *CoRR*, vol. cs.CL/0112018, 2001. [Online]. Available: https://arxiv.org/abs/cs/0112018

[8] A. Abboud, A. Backurs, and V. V. Williams, "If the current clique algorithms are optimal, so is valiant's parser," 2015.

[9] T. M. Chan, "More algorithms for all-pairs shortest paths in weighted graphs," *SIAM Journal on Computing*, vol. 39, no. 5, pp. 2075–2089, 2010. [Online]. Available: https://doi.org/10.1137/08071990X

[10] W. Rytter, "Fast recognition of pushdown automaton and context-free languages," *Information and Control*, vol. 67, no. 1, pp. 12–22, 1985. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0019995885800243

[11] V. L. Arlazarov, E. A. Dinic, M. A. Kronrod, and I. A. F. zev, "On economical construction of the transitive closure of an oriented graph," *Soviet Mathematics Doklady*, vol. 11, pp. 1209–1210, 1970.

[12] M. Yannakakis, "Graph-theoretic methods in database theory." 01 1990, pp. 230–242.

[13] P. Purdom, "A transitive closure algorithm," *BIT*, vol. 10, pp. 76–94, 03 1970.

[14] R. Diestel, *Graph Theory*, 4th ed., ser. Graduate Texts in Mathematics. Springer, 2010, vol. 173.

[15] H. L. Bodlaender and T. Hagerup, "Parallel algorithms with optimal speedup for bounded treewidth," in *Proceedings 22nd International Colloquium on Automata, Languages and Programming*. Springer-Verlag, 1995, pp. 268–279.

[16] H. L. Bodlaender, "Dynamic programming on graphs with bounded treewidth," in *Automata, Languages and Programming*, T. Lepistö and A. Salomaa, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1988, pp. 105–118.